Optimización de funciones de DSP para procesador con instrucciones de aritmética compleja

Autor: García-Castrillón Triquell, Eduardo

Profesor Director: Piñuel Moreno, Luis

Curso académico: 2007 - 2008

Asignatura: Sistemas Informáticos, Facultad de Informática, Universidad Complutense de Madrid

Prefacio

El objetivo principal del proyecto ha sido adaptar la librería matemática del procesador CoolFlux Complex al nuevo tipo de datos empaquetados "complex".

El trabajo se divide en varios subproyectos, uno para cada función de la librería *mathlib*. La labor ha sido construir un nuevo subproyecto que fuera idéntico al de la función antigua, pero con ficheros fuente actualizados al nuevo tipo de datos y todos los ficheros de configuraciones y tests adaptados para utilizar el nuevo procesador.

Con esto se ha pretendido obtener una librería optimizada que mejore el rendimiento de cada función entre un 100 y un 250 % a costa de perder precisión en los valores tratados. Esta pérdida es de la mitad de bits respecto al original debido al formato empaquetado de los nuevos tipos de datos. Teniendo en cuenta la nueva arquitectura del procesador CFC, las optimizaciones hechas en él para el tipo de datos utilizado permiten estos porcentajes de mejora de rendimiento.

My principal objective in this project has been to adapt the CoolFlux Complex mathematical library to new packed data types known as “complex data types”.

The project is divided into a group of subprojects, each one dedicated to a function from de *mathlib* libary. The main job has been to build a new subproject that would be identical to the one of the old mathematical function, but with updated source files and adapted configuration and test files to the new processor.

With this work, the intention has been to obtain an optimized library with a 100 to 250 % performance increase at the expense of losing precision. This loss represents half of the bits compared to the original data types due to the new format of the packed data types. Thanks to the CoolFlux Complex's new architecture, all this performance increases have been possible with the new data types.

Lista de palabras

optimización, CoolFlux, datos complejos, funciones matemáticas, DSP, consumo de energía, arquitecturas especializadas.

Yo, Eduardo García-Castrillón Triquell, autorizo a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, la memoria desarrollada en este proyecto.

Madrid, Junio de 2008

Fdo. Eduardo García-Castrillón Triquell

Agradecimientos

Primeramente, me gustaría dar las gracias a Don Luis Piñuel Moreno, director de este proyecto, por haberme brindado esta gran oportunidad que repercutirá en mi futuro como Ingeniero Informático.

También agradezco el trato recibido por Don Francisco Barat Quesada, quien me ha guiado a lo largo del proyecto y me ha enseñado a desenvolverme en un ambiente de empresa.

Por último me gustaría mostrar mi agradecimiento al resto de miembros del equipo de desarrollo del procesador CoolFlux y de sus librerías, con los que he trabajado: Ms. Marleen Ade y Mr. Jeroen Coninx.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Dentro de la informática existe un campo dedicado al procesamiento digital de señales (PDS). Este procesamiento se realiza mediante el uso de los llamados *Procesadores de Señales Digitales* o DSP (*Digital Signal Processor*).

Una definición informal de procesamiento digital de señales podría ser la aplicación de operaciones matemáticas para representar digitalmente señales. Las señales se representan digitalmente como secuencias de *muestras*. Normalmente, estas muestras se obtienen de señales físicas (por ejemplo, señales de sonido) mediante el uso de transductores[1] (como por ejemplo un micrófono) y convertidores analógico-digital. Después del procesamiento matemático, las señales digitales pueden ser de nuevo convertidas a señales físicas vía un convertidor digital-analógico.

En algunos sistemas, el DSP es el núcleo central de operación. Por ejemplo, los modems y los teléfonos móviles dependen totalmente de la tecnología DSP. En otros productos, el uso de esta tecnología no es vital, pero ofrece ventajas importantes en términos de cualidades, rendimiento y coste. Por ejemplo, los fabricantes de amplificadores de sonido utilizan la tecnología DSP para conseguir nuevas características.

[1] Un transductor es un dispositivo capaz de transformar o convertir un determinado tipo de energía de entrada, en otra diferente de salida. Es un dispositivo usado principalmente en la industria, en la medicina, en la agricultura, en robótica, en aeronáutica, etc. para obtener la información de entornos físicos y químicos y conseguir (a partir de esta información) señales o impulsos eléctricos, o viceversa.

1.1. Ventajas de los DSP

El procesamiento digital de señales alberga muchas ventajas sobre el procesamiento analógico de señales. La más importante de estas ventajas podría ser la **capacidad de realizar operaciones de forma sencilla** que serían muy difíciles o incluso imposibles para la electrónica analógica. Ejemplos de estas aplicaciones podrían ser el reconocimiento de voz, el análisis de voz o modems de alta velocidad con uso de corrección de errores. Todas estas tareas conllevan una combinación de procesamiento de señal y control (como podría ser las decisiones a tomar con la llegada de bits o de voz) que son extremadamente difíciles de implementar usando técnicas analógicas.

Además, los sistemas DSP (de ahora en adelante "DSP") tienen dos ventajas adicionales sobre los sistemas analógicos:

- **Insensibilidad al entorno**: Los sistemas digitales, por su naturaleza, son considerablemente menos sensibles a las condiciones del entorno que los sistemas analógicos. Por ejemplo, el comportamiento de un circuito analógico depende de su temperatura. Sin embargo, sin tener en cuenta fallos de tipo catastrófico, un DSP no depende de su entorno: sea en la nieve o en el desierto, un DSP funciona igual.

- **Insensibilidad a tolerancias de sus componentes**: Los componentes analógicos se fabrican con unas tolerancias determinadas: una resistencia, por ejemplo, puede tener garantizada una resistencia entorno al 1 % de su valor nominal. La respuesta total de un sistema analógico depende de los valores de todos sus componentes analógicos. Como resultado, dos sistemas analógicos con exactamente el mismo diseño tendrán respuestas ligeramente diferentes debido a pequeñas variaciones de sus componentes. Sin embargo, componentes digitales funcionales siempre producirán las mismas salidas dadas las mismas entradas.

Estas dos ventajas se combinan para dar a los DSP otra ventaja adicional sobre los sistemas analógicos:

- **Comportamiento predecible y repetible**: Dado que la salida de un DSP no cambia debido a factores del entorno o variaciones en sus componentes, es posible diseñar sistemas con respuestas exactas y conocidas que no variarán.

Finalmente, algunos DSP pueden tener otras dos ventajas sobre los sistemas analógicos:

- **Reprogramabilidad**: Si un DSP está basado en un procesador programable, éste puede ser reprogramado para realizar otras tareas. En contraste, los sistemas analógicos requieren físicamente componentes nuevos para realizar nuevas tareas.

- **Tamaño**: El tamaño de los componentes analógicos varía en función de sus valores: por ejemplo, un condensador de 100 μF utilizado en un filtro analógico es físicamente más grande que otro de 10 pF utilizado en otro filtro. Sin embargo, la implementación DSP de ambos filtros sería del mismo tamaño, de hecho incluso usaría el mismo hardware, siendo tan sólo distintos los coeficientes del filtro, y seguramente sería más pequeño que cualquiera de las dos implementaciones analógicas.

Estas ventajas, unidas al hecho de que la tecnología DSP puede aprovechar el rápido crecimiento de la densidad del proceso de fabricación de los circuitos integrados, hace que cada vez sea una técnica más utilizada a la hora de procesar señales.

1.2. Características de los DSP

En esta sección se describen algunas características comunes a todos los DSP, como algoritmos, frecuencias de muestreo, frecuencia de reloj y tipos aritméticos.

Algoritmos

Los DSP suelen caracterizarse por los algoritmos que utilizan. El algoritmo especifica las operaciones aritméticas que se realizarán, pero no cómo éstas deben ser implementadas. Pueden ser implementadas en software en un procesador de propósito general o un procesador de señales programable, o incluso implementadas en un circuito integrado propio. La elección de la tecnología de implementación se determina en parte por la velocidad requerida y por la precisión aritmética necesaria. En la tabla 1.1 se muestran algunos tipos comunes de algoritmos para DSP y algunas de sus aplicaciones.

Cuadro 1.1: Algoritmos DSP típicos

Algoritmo	**Aplicación**
Codificación y decodificación de voz	Teléfonos móviles, inalámbricos, comunicaciones seguras
Reconocimiento de voz	Robótica, aplicaciones de automóviles, teléfonos móviles, sistemas de comunicación personal
Codificación y decodificación de alta fidelidad	Sistemas audio, video, audio profesional, emisión de radio digital
Algoritmos de modems	Teléfonos móviles, PDAs, señal digital en televisión por cable, computación inalámbrica
Cancelación de ruido	Audio profesional, audio avanzado para vehículos
Ecualización de audio	audio a nivel de consumidor o profesional, música
Visión	Seguridad, ordenadores multimedia, instrumentación, robótica
Compresión y descompresión de imágenes	Fotografía digital, video digital, video-sobre-voz (videoconferencia)

Frecuencia de Muestreo

La frecuencia de muestreo es una característica clave en los DSP: la frecuencia a la que las muestras son consumidas, procesadas o producidas. Combinada con la complejidad de los algoritmos, la frecuencia de muestreo determina la velocidad de la tecnología de implementación. Un ejemplo es el reproductor de discos compactos de audio (CD), que produce muestras a una frecuencia de 44.1 kHz en dos canales.

Por supuesto, un DSP puede procesar más de una frecuencia de muestreo; a esos sistemas se les denomina *Sistemas DSP Multifrecuencia*. Un ejemplo de estos últimos podría ser el conversor de muestra de CD a 44.1 kHz a la frecuencia de cinta digital (DAP:*Digital Audio Tape*), que es 48 kHz. Debido a lo incómodo de este ratio entre estas dos frecuencias, la conversión suele hacerse en fases, típicamente con al menos dos frecuencias intermedias. Otro ejemplo de multifrecuencias podría ser un banco de filtros, utilizado en aplicaciones como la codificación de voz, audio o video. Los bancos de filtros suelen consistir en etapas que dividen la señal en porciones de alta y baja frecuencias. Estas nuevas señales son *submuestreadas*[2] y divididas de nuevo. En aplicaciones multifrecuencia, el ratio entre la muestra de frecuencia más alta y más baja puede ser muy grande, a veces sobrepasando 100,000.

El rango de muestreo que encontramos en los sistemas de procesamiento de señales es enorme. En la figura 1.1 se muestra el posicionamiento aproximado de algunas clases de aplicaciones con respecto a la complejidad de su algoritmo y la frecuencia de muestreo. Solamente a partir de frecuencias realmente altas la implementación digital no es común. Esto es debido a que el coste y la dificultad de implementación de un algoritmo suele crecer en función de su frecuencia de muestreo. Los algoritmos DSP utilizados en frecuencias altas tienden a ser más simples que los de frecuencias bajas.

Frecuencias de reloj

Los sistemas digitales electrónicos suelen estar caracterizados por su frecuencia de reloj. La frecuencia de reloj se refiere a la frecuencia a la cual el sistema realiza su unidad de trabajo más básica. En productos comerciales de producción en masa, frecuencias de reloj de hasta 100 MHz son normales,

[2]Submuestrear es el proceso de reducir la frecuencia de muestreo de una señal. Suele hacerse para reducir la frecuencia de datos o el tamaño de estos.

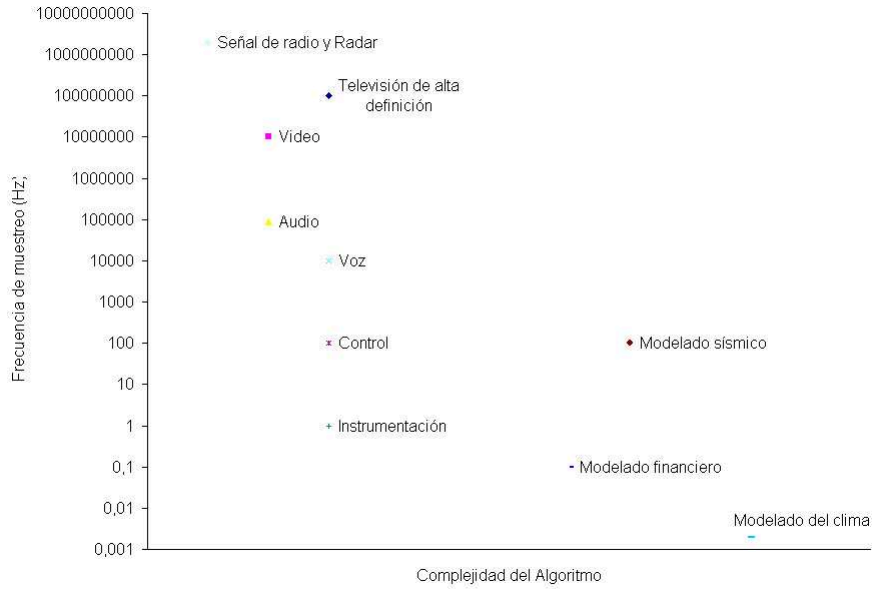

Figura 1.1: Ilustración aproximada de frecuencias de muestreo y complejidad de algoritmos para ciertas clases de aplicaciones de los DSP.

con frecuencias más altas en algunos productos de alto rendimiento. En DSP, el cociente o ratio entre la frecuencia de reloj y la frecuencia de muestreo es una de las características más importantes utilizadas para determinar cuál será la implementación del sistema. La relación entre la frecuencia de reloj y la de muestreo determina en parte la cantidad de hardware necesario para implementar un algoritmo dado con una complejidad dada en tiempo real. Mientras el ratio frecuencia de muestreo / frecuencia de reloj crece, también lo hace la complejidad del hardware necesario para implementar el algoritmo. Es decir, mientras más trabajo le pidamos al DSP por unidad de tiempo (mayor eficiencia) y menor frecuencia de reloj le demos (menor consumo), más complejo será de implementar.

Representaciones Numéricas

Las operaciones numéricas como la adición y multiplicación son el núcleo de los algoritmos de DSP. Como resultado, las representaciones numéricas y el tipo de aritmética utilizadas pueden tener una gran influencia en el comportamiento y rendimiento de un DSP. La decisión más importante para un diseñador se da entre el uso de punto fijo o punto flotante. La aritmética en punto fijo representa números en un rango fijo (por ejemplo, entre -1.0 y +1.0) con un número finito de bits de precisión (llamados *ancho de palabra*).

Por ejemplo, un número en punto fijo con 8 bits obtiene una resolución de 1/256 sobre el rango en el cual el número puede variar. Los números fuera de ese rango no podrán representarse; por lo tanto, las operaciones que como resultado den un número fuera del rango serán o bien *saturadas*, es decir, limitadas al número más alto positivo o negativo representable, o *truncadas*, es decir, los bits extra de la operación serán descartados.

Por otro lado, la aritmética en punto flotante expande de gran manera el rango de valores representable. Esta aritmética representa cada número en dos partes: una *mantisa* y un *exponente*. La mantisa es, en efecto, forzada a entrar en el rango -1.0 y +1.0, mientras que el exponente mantiene la cantidad a la cual la mantisa debe ser escalada (en potencias de 2) para poder crear el valor actual a representar. Es decir,

$$valor = mantisa * 2^{exponente} \tag{1.1}$$

La aritmética en punto flotante provee un rango dinámico mucho mayor que en punto fijo. Dado que ello reduce la probabilidad de *overflow* y la necesidad de escalado, puede simplificar considerablemente el algoritmo y el diseño del software. Pero por otro lado, esta aritmética es generalmente más lenta y más complicada de implementar en hardware. En este proyecto, la aritmética utilizada en el desarrollo del procesador ha sido en punto fijo, dada su simplicidad de implementación y uso.

1.3. Objetivos y Esquema del Proyecto

De entre todos los procesadores de señales actuales, el CoolFlux DSP es el de menor consumo de potencia. Ello hace que sea un procesador muy indicado para aparatos portátiles o que precisen una larga autonomía. Debido a ese objetivo, el uso que debe hacerse del procesador debe ser lo más eficiente posible para minimizar la ejecución de operaciones innecesarias y maximizar así la duración de la batería que lo alimente.

Lo que pretendía hacer en este Proyecto de Fin de Carrera era optimizar una serie de algoritmos aprovechando unos nuevos tipos de datos en una nueva versión del procesador, la CFC (CoolFlux Complex). Esta nueva versión

del procesador es capaz de operar con tipos de datos empaquetados, incluyendo optimizaciones explícitas para números complejos, lo que ha hecho posible la mejora de funciones hasta en un 250 %.

El esquema básico de esta memoria será:

- En el capítulo 2 se describirá el estado de la cuestión, donde se nombrarán algunos campos actuales donde los procesadores digitales de señales son de gran ayuda.
- En el capítulo 3 se encuentra una descripción de las tareas realizadas, así como las fases de cada subproyecto y la duración total del proyecto.
- El capítulo 4 expone el trabajo realizado desde el alto nivel hasta la descripción específica de los módulos generados.
- En el capítulo 5 se recogen los resultados y conclusiones obtenidos durante los cuatro meses de duración del proyecto así como algunas posibles vías de ampliación del trabajo de forma que pueda resolver más problemas o ser más eficiente.
- Finalmente la memoria termina con la mención a la bibliografía utilizada.

Capítulo 2

Estado de la cuestión

2.1. Introducción

Actualmente el campo de los procesadores de señales digitales (DSP) consigue un enorme rendimiento. Esto es debido en parte al gran avance en las arquitecturas y tecnologías como el continuo empequeñecimiento de los módulos, memorias caché de dos niveles y acceso rápido o el ensanchamiento del bus. Aun así, no todos los DSP ofrecen la misma velocidad, ya que existen muchos tipos de procesadores, cada uno de ellos destinado a tareas específicas. Por ello, los precios de los DSP pueden variar desde 1 euro hasta más de 300.

Actualmente los DSP pueden tener frecuencias de reloj de más de 1 GHz, como el Texas Instruments C6000, tener cachés de datos y de instrucciones separadas, varios megabytes de caché de nivel 2 o varios canales DMA. Los DSP actuales son capaces de operar más de 8000 MIPS (millones de instrucciones por segundo), usar VLIW (Very Long Instruction Word), realizar ocho operaciones por ciclo y son compatibles con varios periféricos externos y varios tipos de bus (PCI, serial, etc.).

Los DSP también están especializados en el campo multimedia, como son los codecs (codificador - decodificador), filtros y convertidores digital - analógico. Para ello los diseñadores pueden incluir varios multiplicadores y ALUs en el propio procesador, instrucciones SIMD (Simple Input Multiple Data) y componentes específicos para el procesamiento de sonido.

Otro uso de los DSP es el de los DSP embebidos, que combinan las características de los DSP con las de los procesadores de uso general. Con ello se

consigue que los procesadores sean capaces de ejecutar sistemas operativos simples como μCLinux (Micro Controller Linux), velOSity o Nucleus RTOS, que componen la base de varios productos como routers, cámaras de seguridad, DVD, reproductores MP3, teléfonos VoIP (Voice over IP), scanners o lectores de tarjetas.

Actualmente, los procesadores de uso general utilizan conceptos provenientes de los DSP, como pueden ser varios sets de instrucciones presentes en las extensiones MMX y SSE de Intel.

2.2. El CoolFlux DSP

NXP Semiconductors comenzó siendo una empresa perteneciente a Philips, pero desde el 29 de Septiembre de 2006, tan sólo el 20 % es de Philips, siendo el 80 % restante de inversores privados. Uno de sus productos más importantes ha sido el CoolFlux DSP, un DSP pensado para aplicaciones de ultra bajo consumo. Basado en varios años de experiencia en el diseño de electrónica de ultra bajo consumo, el CoolFlux fue desarrollado para aplicaciones audio portátiles, incluyendo codificación y decodificación de sonido, algoritmos para la mejora de sonido y supresión de ruido. Los productos a los que se enfoca esta tecnología incluye auriculares, audífonos y reproductores portátiles de audio.

Beneficios

Las ventajas que ofrece el CoolFlux DSP sobre otros DSP de bajo consumo son:

- Ultra bajo consumo:
  - 25 μW/MHz a 0.9 V (CMOS[1] de 65 nm)
  - 45 μW/MHz a 1.2 V (CMOS de 65 nm)
  - 34 μW/MHz a 0.9 V (CMOS de 90 nm)

[1]Del inglés *Complementary Metal-Oxide Semiconductor*, es un tipo de semiconductor ampliamente utilizado que usa circuitos tanto NMOS (polaridad negativa) como PMOS (polaridad positiva). Dado que solamente uno de los dos circuitos puede estar activo en un momento dado, los chips CMOS requieren menos energía que los chips que utilizan tan sólo un tipo de transistor.

  - 60 μW/MHz a 1.2 V (CMOS de 90 nm)
  - 45 μW/MHz a 0.9 V (CMOS de 130 nm)
  - 80 μW/MHz a 1.2 V (CMOS de 130 nm)
  - 60 μW/MHz a 0.9 V (CMOS de 180 nm)
  - 240 μW/MHz a 1.8 V (CMOS de 180 nm)
- Kit de desarrollo software basado en un compilador de C altamente optimizado
- Tamaño mínimo del core de 43.000 puertas
- Rendimiento:
  - 310 MHz WCCOM a 1.2 V con CMOS de 65 nm (>2300 MOPS[2])
  - 250 MHz WCCOM a 1.2 V con CMOS de 90 nm
  - 208 MHz WCCOM a 1.2 V con CMOS de 130 nm
  - 168 MHz WCCOM a 1.8 V con CMOS de 180 nm (>1000 MOPS)
- Una extensa librería de software de decodificación de audio y algoritmos avanzados de sonido

[2]Del inglés *Millions of Operations per Second*, en español *Millones de Operaciones por Segundo*

Capítulo 3

Gestión del proyecto

El trabajo ha consistido en optimizar varias funciones matemáticas, lo que permite la división del conjunto del proyecto de manera fácil.

Cada una de las funciones que han sido tratadas se puede tomar como paquete de trabajo individual o subproyecto del proyecto global, a los que llamaré también proyectos. Cada uno de estos, salvo por alguna comparación de ficheros, es independiente de las demás. A continuación expongo la división del trabajo así como la duración que ha conllevado cada uno de los proyectos:

Paquetes de trabajo y duración:

1. IOlib - 2 semanas
2. Fir - 3 semanas
3. BiQuad - 2 semanas
4. BiQuad dpfb - 1 semana
5. Vector operation - 2 días
6. FFT Complex - 6 semanas

Para procesar cada uno de estos paquetes, he seguido una serie de fases de forma rutinaria:

Fase 1: familiarización con el proyecto de trabajo.
Fase 2: adaptación del proyecto al nuevo procesador.
Fase 3: estudio de viabilidad para la actualización.

Fase 4: implementación de la actualización en un nuevo proyecto de trabajo.
Fase 5: pruebas y validación.

- El objetivo de la fase 1 ha sido la obtención de los ficheros del proyecto y lecturas de los documentos incluidos en éste. Los ficheros incluyen proyectos completos para Visual Studio 6, 7 y 8, proyecto para ChessDE (el compilador de CFC), pruebas de validación mediante ficheros *batch*[1] y un ejemplo de ejecución completo tanto para el modo Win32 nativo como el modo CoolFlux.
- En la fase 2 se modifican los ficheros fuente y las baterías de pruebas para que utilicen el nuevo procesador CFC en lugar de CF6 y se comprueba que los resultados obtenidos sean los mismos que con el procesador anterior.
- La fase 3 consiste en la comprensión de la función para poder aplicar una optimización en el código fuente que permita ganancias en el rendimiento considerables. Debido a que el nuevo tipo de datos trabaja con la mitad de precisión, las ganancias en velocidad deben ser altas para que se considere viable.
- En la fase 4 se crea un nuevo proyecto implementando las posibles mejoras obtenidas en la fase 3 para ambas plataformas, Win32 y CoolFlux. A pesar de ser dos plataformas, casi la totalidad del código fuente C es compatible para ambas. Las partes no compatibles se separan mediante las definiciones de preprocesado:

  ```
  #ifdef Win32

  // código solo compatible con Win32

  #endif
  ```

  y

  ```
  #ifdef CoolFlux
  ```

[1] En Microsoft Windows, un fichero batch es un fichero de texto que contiene una serie de comandos que ejecutará el intérprete de comandos. Cuando es ejecutado, un fichero batch se suele procesar línea por línea. Suelen utilizarse para automatizar procesos tediosos.

```
// código solo compatible con CoolFlux

#endif
```

- Finalmente, la fase 5 comprende dos partes: comprobar que los resultados de las pruebas sean idénticos tanto para CoolFlux como para nativo (Win32) y validar las nuevas funciones. Esta última parte es la que más tiempo suele conllevar debido a que los resultados de las pruebas no son los mismos que los del proyecto anterior de CF6. Por ello, hay que reescribir los resultados manualmente o crear una batería de pruebas distinta que verifique exhaustivamente todas las funciones nuevas.

Capítulo 4

Trabajo Realizado

Antes de comenzar a relatar el trabajo realizado, voy a hacer primero una introducción al procesador CoolFlux Complex para poder entender los objetivos.

4.1. Visión general

4.1.1. Arquitectura del CoolFlux Complex

El CoolFlux Complex es un DSP de propósito general de alta precisión, consumo ultra bajo, bajo coste, Harvard dual[1], capaz de soportar dos MAC[2], dos operaciones de memoria y dos actualizaciones de puntero por ciclo, haciéndolo altamente eficiente en ciclos para aplicaciones intensivas de computación. Varias de las operaciones aritméticas, como multiplicación, suma, división, etc. pueden hacerse en modo escalar o en modo empaquetado. El modo empaquetado comprende las operaciones complejas y las operaciones SIMD[3]. Trabajando así, el CoolFlux Complex es capaz de realizar cuatro multiplicaciones de 12 bits en paralelo.

[1]La arquitectura Harvard es una arquitectura que separa físicamente el almacenamiento de instrucciones y datos en dos memorias.

[2]MAC viene del ingles multiply accumulate y es el tipo de operación que realiza una multiplicación y añade el resultado a una variable. Por ejemplo: x = x + (y * z)

[3]SIMD es el acrónimo inglés Simple Input Multiple Data.

El CoolFlux Complex tiene alto soporte para aritmética fraccionaria y soporte altamente eficiente para tipos de datos y operaciones de ANSI C.

El CFC extiende la base del CoolFlux anterior mediante soporte hardware para aritmética compleja, paralelismo a nivel de subpalabra y un espacio de direccionamiento mayor.

La arquitectura de este DSP es una arquitectura load/store, esto es:

- Todos los operandos de memoria deben ser movidos explícitamente a registros.
- Todas las unidades de ejecución toman sus operandos de registros y escriben el resultado en registros.
- Todos los resultados producidos en los registros deben moverse explícitamente a memoria.

La arquitectura del CoolFlux Complex puede verse en la figura 4.1. Para una descripción más detallada, a continuación se explica la arquitectura que puede verse en las figuras 4.2 y 4.3.

En el esquema de la figura 4.2, se pueden ver:

- La Unidad de Control de Programa (Program Control Unit) con los registros pc, sr, isr, lr e ilr y la memoria de programa de 16 megapalabras de 32 bits.
- Las unidades de direccionamiento X e Y, que contienen:
  - Las unidades de generación de direcciones XAGU e YAGU.
  - Los registros xptr, xstep, xmod y sp.
  - Los registros yptr, ystep e ymod.
- Las memorias X, Y y IO de 16 megapalabras de 24 bits.
- El bloque DMA.
- Los buses de 24 bits Xbus e Ybus.

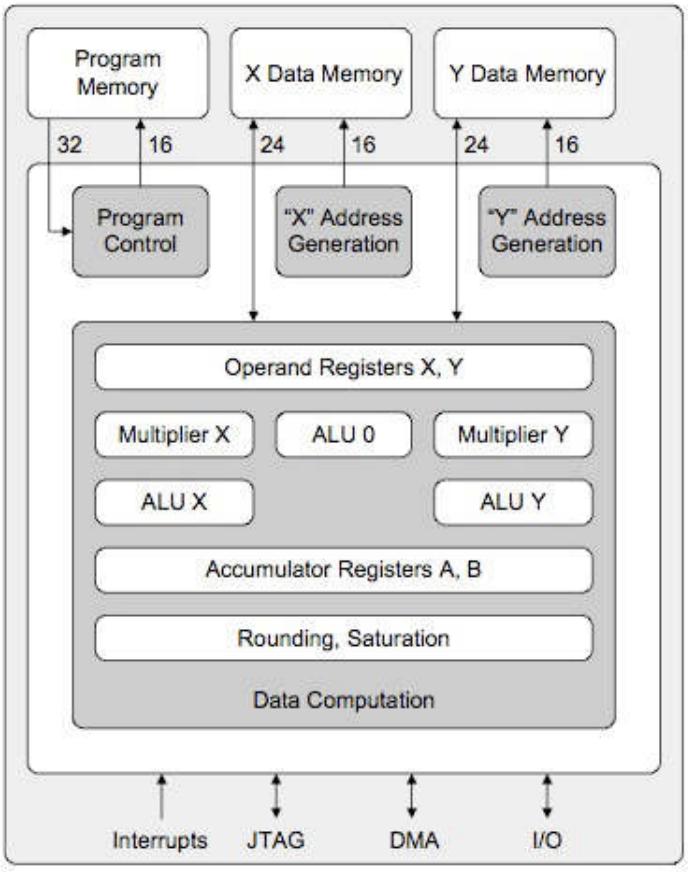

Figura 4.1: Arquitectura del CoolFlux Complex

Y en el esquema de la figura 4.3 aparecen:

- Los archivos de registro X e Y con los registros de 24 bits x0, x1, y0, y1.
- Los archivos de acumuladores A y B con los registros de 56 bits a0, a1, b0, b1.
- La vía de datos, con:
  - Los multiplicadores Xmul e Ymul.
  - La unidad de pre-suma/resta de Xmul.
  - Las ALUs XALU e YALU.
- Las dos unidades de redondeo, saturación y selección RSS (Round, Saturate and Select).

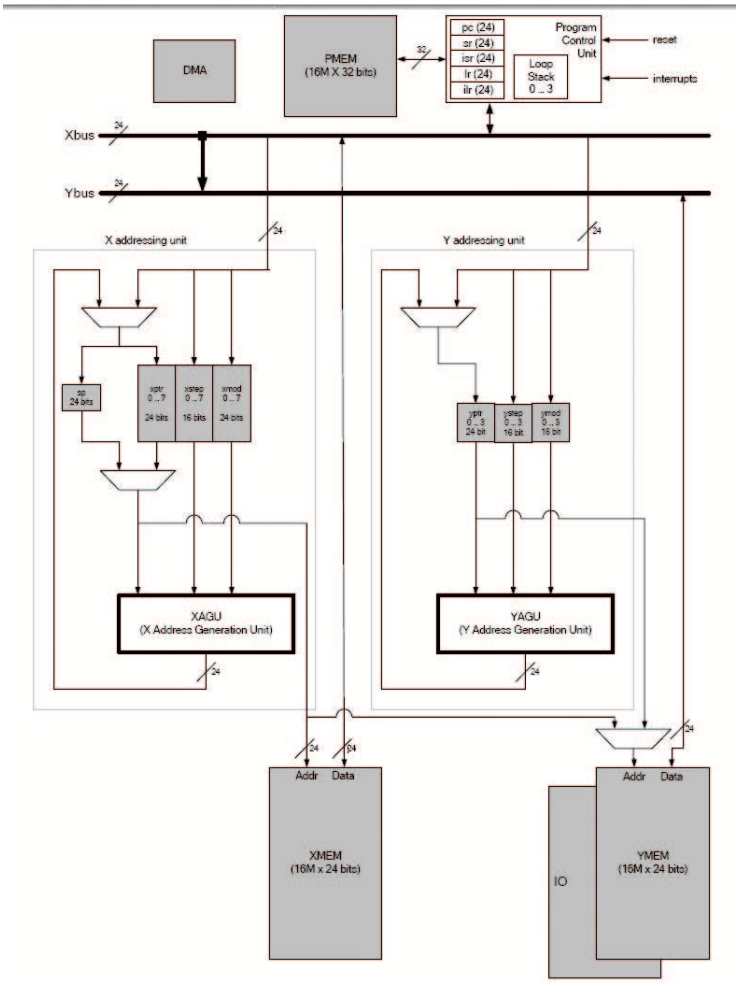

Figura 4.2: CoolFlux Complex: Datapath I

- Los bloques de conversión de datos de diferente tamaño.

4.1.2. Tipos de datos

Los tipos de datos que soporta CoolFlux Complex son *int, int24, uint* (*unsigned int*), *uint24, long, ulong, long48, ulong48, fix, lfix, acc, complex12, complex24, complex28, simd12, simd24* y *simd28*. Estos seis últimos son los nuevos tipos empaquetados que no estaban presentes en el anterior CFC.

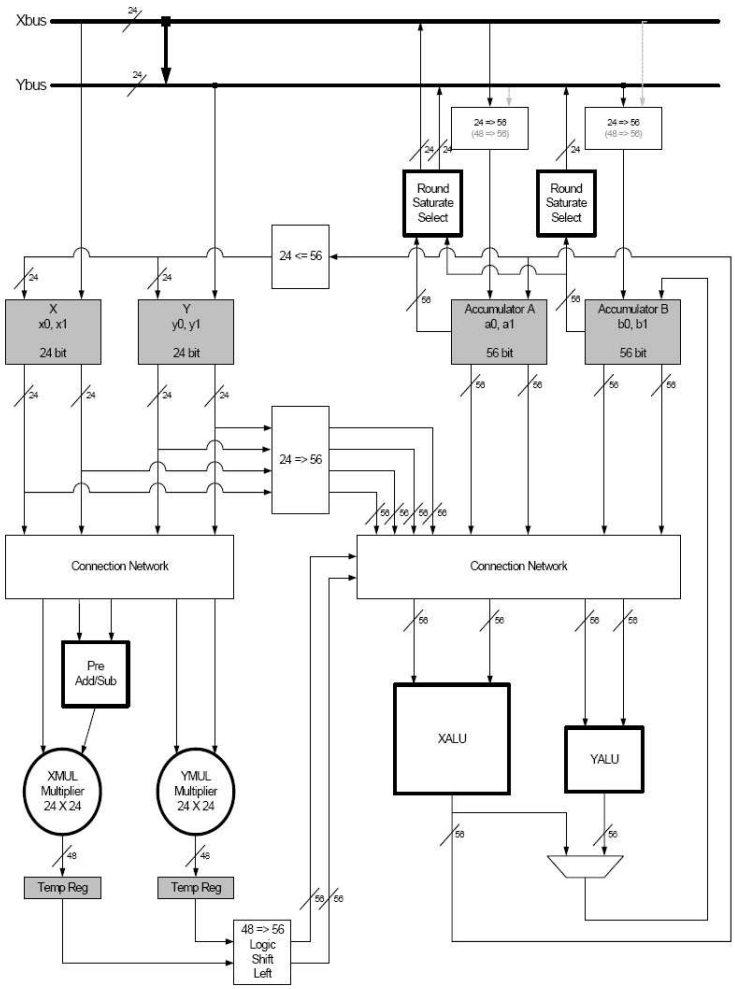

Figura 4.3: CoolFlux Complex: Datapath II

En el cuadro “Tipos escalares” se puede ver información acerca de los tipos de datos escalares, mientras que para los nuevos tipos de datos, los empaquetados, la información es la que aparece en la tabla “Tipos empaquetados”.

- *int24* es la versión de 24 bits de *int* necesaria en situaciones que tratan señales de audio de 24 bits.

- *long* es un tipo de 48 bits en CoolFlux, pero debido a que en la plataforma C el número de bits es 32 (*long int*), se proporciona el tipo *long48* para la parte nativa de Win32.

- Los tipos *fix* y *lfix* representan números con coma fija después del bit más significativo.

- El tipo *acc* es la versión de 56 bits de *fix* y está dividido en tres partes: 8 bits de overflow, 24 bits para una palabra (*high word*) y 24 para otra (*low word*).

Cuadro 4.1: Tipos escalares

Nombre	Número de bits	Tipo	Rango
int, int24	24	entero	$[-2^{23}, 2^{23}-1]$
unsigned int, uint, uint24	24	entero	$[0, 2^{24}-1]$
long, long48	48	entero	$[-2^{47}, 2^{47}-1]$
unsigned long, ulong, ulong48	48	entero	$[0, 2^{48}-1]$
fix	24	punto fijo	$[-1, 1-2^{-23}]$
lfix	48	punto fijo	$[-1, 1-2^{-47}]$
acc	56	punto fijo	$[-2^{8}, 2^{8}-2^{-47}]$

Cuadro 4.2: Tipos empaquetados

Nombre	Tipo	Bits	Bits por componente	Rango por componente
complex12	complex	24	12	$[-1, 1-2^{-11}]$
complex24	complex	48	24	$[-1, 1-2^{-23}]$
complex28	complex	56	28	$[-2^{3}, 2^{3}-2^{-23}]$
simd12	SIMD	24	12	$[-1, 1-2^{-11}]$
simd24	SIMD	48	24	$[-1, 1-2^{-23}]$
simd28	SIMD	56	28	$[-2^{3}, 2^{3}-2^{-23}]$

La estructura de estos nuevos tipos de datos se muestra en las figuras 4.4 y 4.5, que separa en celdas cada bit y apunta dónde está la coma, ya que los números con que trabaja son de punto fijo.

El tipo de datos complex es con el que he trabajado en mayor profundidad. Por ello me dedicaré exclusivamente a hablar de él, aunque a menudo lo compararé con otros tipos de datos del procesador.

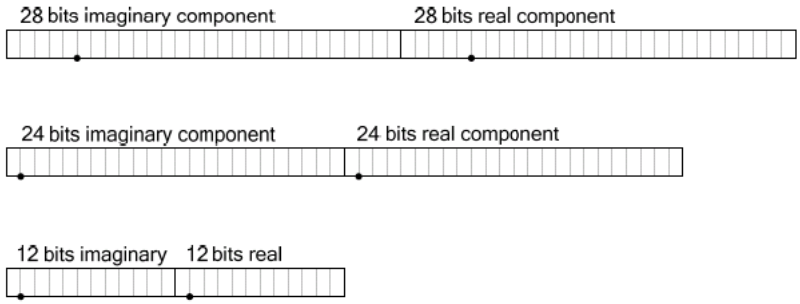

Figura 4.4: Tipos de datos Complex

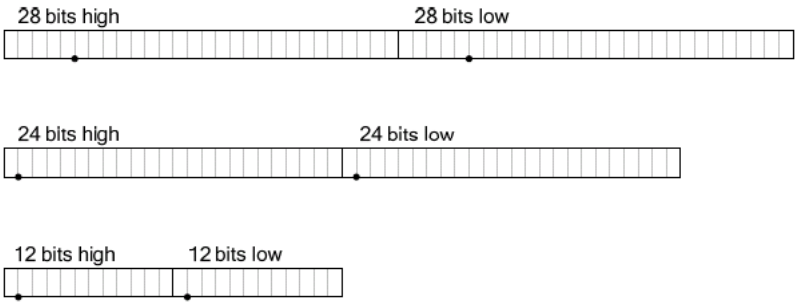

Figura 4.5: Tipos de datos SIMD

Como se puede apreciar en las figuras, cada variable de los tipos de datos empaquetados (complex y simd) necesita el doble de espacio para la misma precisión ya que alberga una parte real y otra imaginaria (para los tipos complex) o una parte alta y una baja (para simd). Por ello, utilizar estos tipos de datos penaliza la precisión de los resultados, pero como el CoolFlux Complex ha sido optimizado para estos nuevos tipos, las operaciones se realizan con un nivel de paralelismo que repercute en una gran mejora en el rendimiento.

4.2. Trabajo realizado

A continuación describo el trabajo realizado a lo largo de los cuatro meses de prácticas en NXP Lovaina, Bélgica. Como se describió en el capítulo anterior, dividiré en paquetes de trabajo el resto del capitulo para una mejor organización:

1. Librería IOlib
2. Filtro FIR
3. Filtro BiQuad
4. Filtro BiQuad de doble precisión
5. Vector Operation
6. Transformada rápida de Fourier compleja

La librería IOlib es la librería de entrada salida que utilizan todos los demás proyectos CoolFlux para realizar funciones de escritura y lectura en memoria. El resto de proyectos son filtros y funciones de transformación matemáticos que han obtenido mejoras en el rendimiento con el uso del nuevo tipo de datos complex.

4.2.1. Librería IOLib

La librería IOlib contiene las operaciones necesarias para poder acceder a la memoria de CoolFlux Complex mediante funciones de lectura y escritura. Cada tipo de datos del procesador tiene las mismas operaciones disponibles, que son:

WIN32

```
void write_tipodato_array(FILE * fptr, tipodato * arr, int length);

void read_tipodato_array(FILE * fptr, tipodato * arr, int length);

void write_tipodato_scalar(FILE * fptr, tipodato scalar);
tipodato read_tipodato_scalar(FILE * fptr);
```

```
void bwrite_tipodato_array(FILE * fptr, tipodato * arr, int length);

void bread_tipodato_array(FILE * fptr, tipodato * arr, int length);

void bwrite_tipodato_scalar(FILE * fptr, tipodato scalar);
tipodato bread_tipodato_scalar(FILE * fptr);
```

CoolFlux

```
void write_tipodato_array(volatile tipodato chess_storage(IOMEM) * cfcvar,
                          tipodato * arr,
                          int length);

void write_tipodato_array(volatile tipodato chess_storage(IOMEM) * cfcvar,
                          tipodato chess_storage(YMEM) *  arr,
                          int length);

void read_tipodato_array(volatile tipodato chess_storage(IOMEM) * cfcvar,
                         tipodato * arr,
                         int length);

void read_tipodato_array(volatile tipodato chess_storage(IOMEM) * cfcvar,
                         tipodato chess_storage(YMEM) * arr,
                         int length);

void write_tipodato_scalar(volatile tipodato chess_storage(IOMEM) * cfcvar,
                           tipodato scalar);

tipodato read_tipodato_scalar(volatile tipodato chess_storage(IOMEM) * cfcvar);
```

En el lugar donde aparece *tipodato* se debe especificar el tipo de datos con el que se trabaja, es decir, *complex12, complex24* o *complex28* en el caso de los números complejos.

Las ocho primeras funciones son las que se utilizan en el modo nativo de Win32 y las seis siguientes en el modo CoolFlux.

Las funciones en las que aparece el término *scalar* son para leer o escribir un valor en memoria. Si aparece el término *array* son para guardar una lista de valores.

En Win32, las funciones que empiezan por la letra *b* son las que se utilizan para escribir en ficheros.

```
/* complex28 */
void write_complex28_array(FILE * fptr, complex28 * arr, int length)
{
    int i;
    int ovf, high, low;
    acc value;

    for(i=0;i<length;i++)
    {

        value = to_acc(arr[i]);

        ovf = extract_ovf(value);
        high = to_int(extract_high(value));
        low = to_int(extract_low(value));

        WRITE(fptr, "%d\n", ovf);
        WRITE(fptr, "%d\n", high);
        WRITE(fptr, "%d\n", low);
    }
}
```

Figura 4.6: Un ejemplo: Código para la función *write* del tipo de datos *complex28* para escribir en memoria un valor

En CoolFlux, las funciones con el mismo nombre se diferencian en la memoria de la que procede el dato (especificada en los parámetros de la función), ya que para realizar operaciones de memoria en paralelo, CoolFlux Complex permite declarar variables en la memoria *X* y en la memoria *Y*.

En el caso de los tipos de datos complex, se trataba de hacer las funciones para *complex12, complex24* y *complex28*. La diferencia entre los tres tipos es el número de bits que utilizan. Dado que las palabras que se escriben en memoria siempre son de 24 bits y cada tipo consta de parte real e imaginaria, tan sólo los datos en *complex12* pueden guardarse en una palabra porque utilizan 12 bits para la parte real y 12 para la imaginaria. Por ello, en los tipos *complex24* las escrituras en memoria se deben guardar en dos palabras. Aun así, la separación de una variable *complex24* es sencilla dadas las operaciones *extract_real* y *extract_imag*, que extraen la parte real y la imaginaria del valor respectivamente. El caso de *complex28* es similar, pero la división ha tenido que ser en tres partes debido a los 56 bits que ocupa una variable de este tipo. Pero ello tampoco ha sido complicado gracias a las operaciones *extract_ovf, extract_real* y *extract_imag*. La función *extract_ovf* devuelve los ocho bits de overflow de la variable.

4.2.2. Filtro FIR

La función del filtro FIR es una función que implementa un filtro no recursivo. Cada sección de orden N del filtro FIR se calcula de acuerdo a las siguientes ecuaciones:

En el dominio Z:

$$H(z) = A_0 z^0 + A_1 z^{-1} + ... + A_N z^{-N} \tag{4.1}$$

O en el dominio del tiempo:

$$y(n) = A_0 x(n) + A_1 x(n-1) + A_2 x(n-2) + ... + A_N x(n-N) \tag{4.2}$$

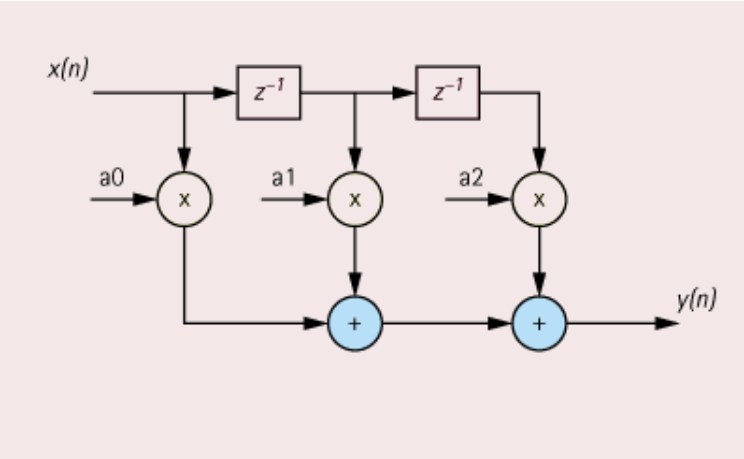

Figura 4.7: Un filtro FIR de 3 etapas

Los coeficientes se guardan en un array en el siguiente orden:

$$A_0, A_1, A_2, ..., A_N \tag{4.3}$$

De esta manera el array contiene N+1 elementos.

Debido al enfoque en el ámbito del filtrado de señales acústicas, el proyecto contiene tres funciones: una versión de muestra mono, una versión basada en bloques y una versión estéreo.

Versión de muestra monoaural:

La función *fir_1ch_sb_Process* computa una un filtro FIR en forma directa usando coeficientes guardados en el vector *pFilterCoeff*. La muestra de entrada se almacena en *in_sample*. El resultado de la aplicación del filtro lo devuelve la propia función. La función retiene la dirección de memoria del retardo anterior en el puntero *ppDelay* para poder llamar a la función en bloques consecutivos.

La función devuelve el valor de la aplicación del filtro y el puntero al retardo módulo buffer es actualizado y queda apuntando al buffer de entrada *x(n)*.

La cabecera de la función es:

```
complex12 fir_1ch_sb_Process(
  complex12                                in_sample,
  const complex12 chess_storage(YMEM)  *pFilterCoeff,
  int                                      FilterLength,
  complex12                                **ppDelay,
  complex12                                *pDelayStart);
```

Donde:

- *in_sample* la muestra de entrada
- *pFilterCoeff* es el puntero a los coeficientes, almacenados en la memoria Y del DSP como se define en la declaración del tipo mediante *chess_storage (YMEM)*
- *filterLength* es la longitud del filtro o numero de coeficientes del filtro
- *ppDelay* es un puntero a un puntero al retardo modulo buffer que apunta a x(n-1). Usa actualización de puntero circular debido al modulo.
- *pDelayStart* es un puntero al comienzo del retardo

```
complex12 fir_1ch_sb_Process(complex12 in_sample, const complex12 chess_storage(YMEM) *pFilterCoeff,
                             int FilterLength, complex12    **ppDelay,  complex12 *pDelayStart)

{

    complex28 sum;
    int i;
    complex12 * pDelay = *ppDelay;

    sum = (in_sample) * (*pFilterCoeff++); // a0 * x(n)

    for(i=1; i < FilterLength ;i++) chess_loop_range(2, ) chess_prepare_for_pipelining
    {
        sum += (*pDelay) * (*pFilterCoeff++); // ai * x(n-i)    pFilterCoeff++
        pDelay = cyclic_add(pDelay,1,pDelayStart,FilterLength-1); // pDelay++ (= (*ppDelay)++)
    }

    pDelay = cyclic_add(pDelay,-1,pDelayStart,FilterLength-1); // pDelay-- (= (*ppDelay)--)

    *pDelay = in_sample; // *ppTap=pDelay -> x(n), ready for next time
    *ppDelay = pDelay;

    return complex12(sum);

}
```

Figura 4.8: Un ejemplo: Código para la función *fir_1ch_sb_process*.

Versión basada en bloques

La función *fir_1ch_bb_Process* computa un filtro FIR en forma directa usando coeficientes guardados en el vector *pFilterCoeff*. Las muestras de entrada están almacenadas en el vector *in_buff*. La salida del filtro estará en el vector *out_buff*. Como en la versión mono, un puntero *ppDelay* mantendrá la dirección de los anteriores retardos para permitir el procesamiento consecutivo de varios bloques.

Con el objetivo de optimizar el código, cada vuelta se procesa dos muestras, lo que hace necesario incluir la precondición de tener un número par de muestras y de tamaño de filtro.

La cabecera de la función es:

```
int fir_1ch_bb_Process(
  complex12                              *in_buff,
  complex12                              *out_buff,
  int                                    B,
  const complex12 chess_storage(YMEM)    *pFilterCoeff,
```

```
  int                                   FilterLength,
  complex12                             **ppDelay
  complex12                             *pDelayStart);
```

Donde:

- *in_buff* son las muestras de entrada
- *out_buff* las muestras filtradas
- B el tamaño del bloque, que debe ser como mínimo 2 debido a la optimización del código.

Los tres parámetros finales son los mismos que en la versión mono.

Versión estéreo

La función *fir_2chil_bb_Process* computa el filtro FIR utilizando los coeficientes guardados en el vector apuntado por *pFilterCoeff*. Las muestras estéreo están almacenadas en el vector *in_buff* de manera entrelazada. El resultado del filtrado estará en el vector *out_buff* también de manera entrelazada. La función mantiene las posiciones de los retardos de cada canal mediante los punteros *ppDelay1* y *ppDelay2*.

La cabecera de la función es:

```
int fir_2chil_bb_Process(
  complex12                             *in_buff,
  complex12                             *out_buff,
  int                                   B,
  const complex12 chess_storage(YMEM)   *in_pFilterCoeff,
  int                                   FilterLength,
  complex12                             **ppDelay1,
  complex12                             **ppDelay2);
```

Los parámetros son los mismos que en la función mono basada en bloques a excepción de los dos punteros al retardo, uno para cada canal.

Rendimiento

A continuación se muestra una tabla con el rendimiento obtenido en ciclos ejecutando las tres funciones en el simulador ISS del compilador Chess para CoolFlux Complex.

Cuadro 4.3: Rendimiento FIR Complex

Función	**Tamaño bloque**	**Tamaño filtro**	**Ciclos por llamada**
fir_1ch_sb_Process	-	Cualquiera	$13 + (Filterlength - 2)$
fir_1ch_bb_Process	Par	Par	$13 + BlockLength/2 * (28 + (FilterLength/2 - 2) * 2)$
		Impar	$13 + BlockLength/2 * (30 + (INT(FilterLength/2) - 2) * 2)$
	Impar	Cualquiera	no soportado(sólo tamaños de bloque pares)
fir_2chil_bb_Process	Par	Cualquiera $>= 3$	$17 + BlockLength$*$(20 + (FilterLength - 2) * 2)$
	Impar	Cualquiera	no soportado (sólo tamaños de bloque pares)

Y estos son los resultados anteriores obtenidos al ejecutar sobre la versión anterior de CoolFlux, el CF6, con los tipos de datos *fix*.

Cuadro 4.4: Rendimiento FIR Fix

Función	**Tamaño bloque**	**Tamaño filtro**	**Ciclos por llamada**
fir_1ch_sb_Process	-	Cualquiera	$13 + (Filterlength - 2)$
fir_1ch_bb_Process	Par	Par	$14 + BlockLength/2 * (27 + (FilterLength/2 - 2) * 2)$
		Impar	$14 + BlockLength/2 * (29 + (INT(FilterLength/2) - 2) * 2)$
	Impar	Cualquiera	no soportado(sólo tamaños de bloque pares)
fir_2chil_bb_Process	Cualquiera	Cualquiera $>= 3$	$17 + BlockLength$*$(18 + (FilterLength - 2) * 2)$

Se puede comprobar que los valores en la versión anterior son menores que en la nueva, pero esto es debido a que la nueva versión no estaba totalmente optimizada para los nuevos tipos de datos introducidos cuando se realizaron las simulaciones.

Pero esta comparativa no es equitativa. Las nuevas funciones que trabajan con datos complex pueden procesar el doble de datos por cada ejecución

debido a la separación en campos de cada variable en parte imaginaria y parte real.

Modificando las funciones antiguas *fix* para que realicen el mismo trabajo que las nuevas funciones complex, los resultados son bien distintos. He realizado una prueba para reflejar dichas diferencias en la que para cada valor de entrada complex le corresponden dos valores *fix*, uno para la parte imaginaria y otro para la parte real. Además, a la hora de realizar una multiplicación, se opera de manera compleja, es decir:

$$(a + bi) * (c + di) = (a * c - b * d) + (b * c + a * d)i \tag{4.4}$$

A continuación muestro dos tablas comparativas con el número de ciclos que tarda en ejecutar cada función dependiendo del número de muestras a analizar, tamaño del filtro (expresado en forma de dos números "a - b" que representan las muestras del filtro y los coeficientes del filtro respectivamente) y función utilizada. La nueva función es la que aparece como *Fix,Fix* debido al uso de dos "fixes" para cada valor.

Cuadro 4.5: FIR Complex vs FIR FixFix

Basado en muestras	**8 - 4**	**8 - 8**	**32 - 16**	**128 - 64**
FixFix	368 (ciclos)	592	4160	59648
Complex	120	152	864	9600
Veces más rápido	*3.07*	*3.89*	*4.81*	*6.21*

Cuadro 4.6: FIR Complex vs FIR FixFix

Basado en bloques de muestras	**8 - 4**	**8 - 8**	**32 - 16**	**128 - 64**
FixFix	462	766	5454	80142
Complex	125	141	653	5645
Veces más rápido	*3.70*	*5.43*	*8.35*	*14.20*

Como se puede ver, esta vez sí se pueden ver las mejoras que introduce el nuevo DSP para el procesamiento de números complejos. De forma gráfica, los resultados obtenidos aparecen en la figura 4.9 para la versión basada en muestras y en la figura 4.10 para la versión basada en bloques de muestras.

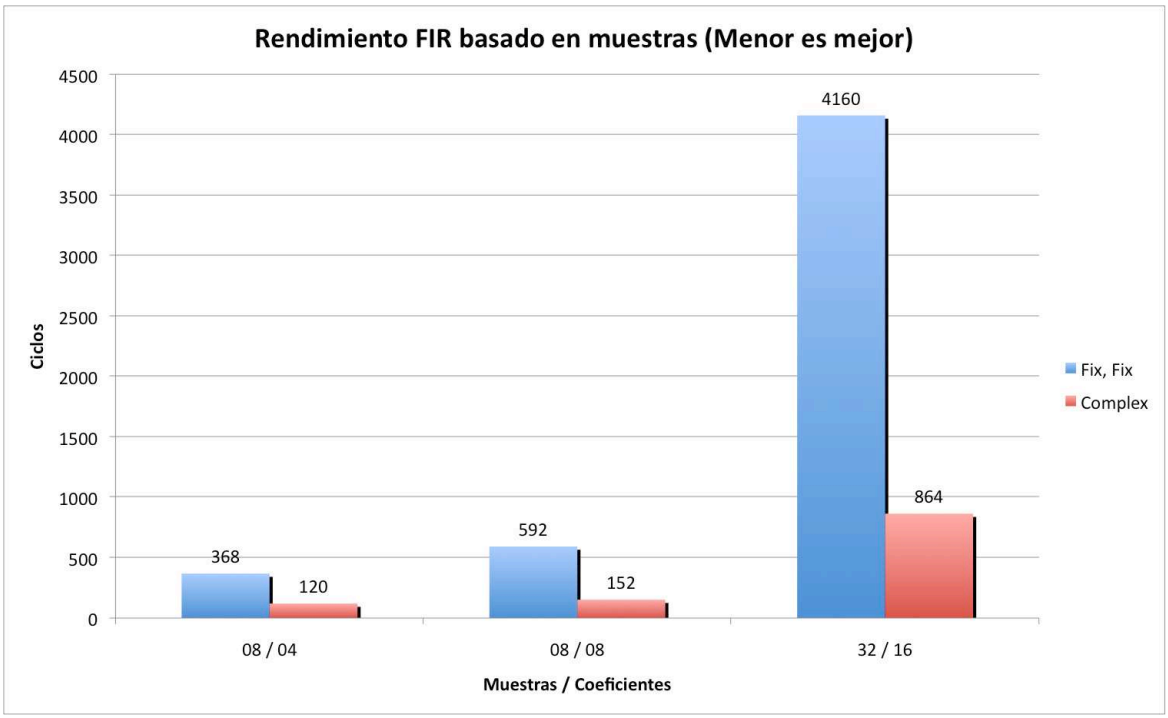

Figura 4.9: Rendimiento FIR Complex procesando muestras

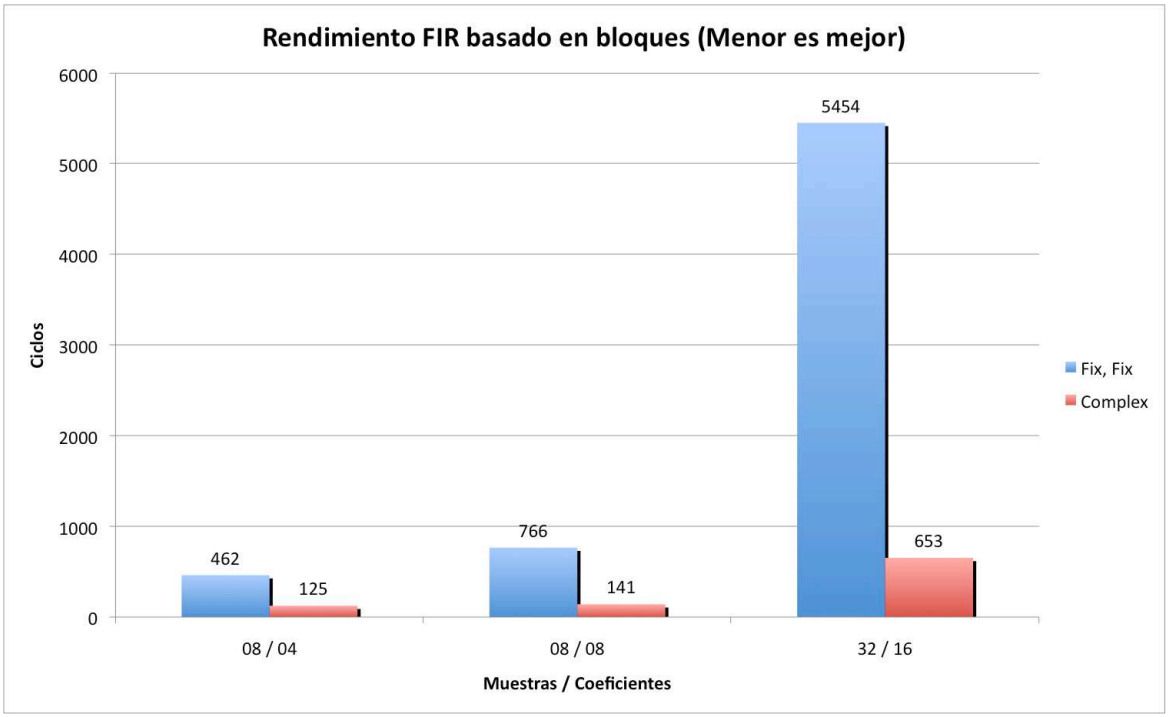

Figura 4.10: Rendimiento FIR Complex procesando bloques de muestras

4.2.3. Filtro BiQuad

Especificación Funcional

Biquad es la abreviación de Filtros Bicuadráticos en Cascada. El tipo de filtro elegido ha sido la Forma Directa I.

La Forma Directa I es preferible a la Forma Directa II (que usa menos coeficientes, retardos y operaciones) porque obtiene mejor precisión y tiene menor riesgo de provocar overflow. En FD-I sólo hay un punto en el algoritmo en el que se realiza una suma contra dos puntos en FD-II. Además, en esta suma, los polos y los ceros se compensan inmediatamente el uno al otro, algo que no ocurre en FD-II, donde hay una adición para los ceros y otra para los polos, siendo esta última una fuente potencial de grandes overflows.

Una sección del filtro Biquad se calcula de acuerdo a las siguientes ecuaciones:

$$H(z) = \frac{(a_0 z^0 + a_1 z^{-1} + a_2 z^{-2})}{(1 + b_1 z^{-1} + b_2 z{-}2)} \tag{4.5}$$

ó

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) - b_1 y(n-1) - b_2 y(n-2) \tag{4.6}$$

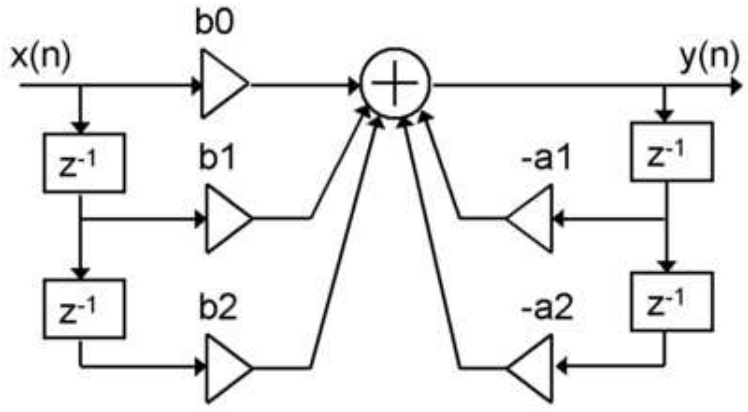

Figura 4.11: Filtro BiQuad

Por esto, una sección de este biquad necesitará:

- 5 coeficientes $(a_0, a_1, a_2, b_1, b_2)$
- 4 elementos en la línea de retardo $(x(n-1), x(n-2), y(n-1), y(n-2))$

El filtro general se calculará como una cascada de N secciones Biquad:

$$H(z) = H_1(z)H_2(z)...H_N(z) \tag{4.7}$$

En el proyecto hay definidas tres funciones:

- Biquad de una sección para señales monoaural con coeficientes no escalados.
- Biquad de N secciones para señales monoaural con coeficientes no escalados.
- Biquad de una sección con coeficientes escalados y punteros en la señal de entrada y de salida que permiten trabajar directamente con señales estéreo entrelazadas.

Biquad de una sección para señales mono

La función *biquad_1ch_bb_1section_Process* procesa datos de un sólo canal con un filtro biquad de una sección. El proceso se realiza en bloques (block based) y en la Forma Directa I. El procesamiento no puede realizarse sobre el vector de datos de entrada, debe haber otro vector de salida, pero ambos pueden ser de longitud par o impar.

La cabecera de la función es:

```
int biquad_1ch_bb_1section_Process (
      complex12                    in_buff,
      complex12                    out_buff,
      int                          B,
      complex12 chess_storage(XMEM) pDelay,
const complex12 chess_storage(YMEM) pFilterCoeff
)
```

Donde:

- *in_buff* es el puntero al vector de entrada de B elementos
- *out_buff* es el puntero al vector de salida de B elementos
- B número de elementos de los vectores *in_buff* y *out_buff*
- *pFilterCoeff* es el puntero al vector de coeficientes, guardados consecutivamente en memoria en orden a_0, a_1, a_2, b_1, b_2
- *pDelay* es el puntero a la línea de retardo. Cada biquad tiene *BIQUAD_DELAYS_PER_SECTION* elementos en la línea de retardo guardados consecutivamente en la memoria en orden:
  $x(n-1)x(n-2)y(n-1)y(n-2)$

Biquad de N secciones para señales mono

La función *biquad_1ch_bb_nsection_Process* es igual al biquad de una sección solo que procesa *num_Sections* secciones en cada llamada.

La cabecera de la función es:

```
int biquad_1ch_bb_nsection_Process (
      complex12                       in_buff,
      complex12                       out_buff,
      int                             B,
      complex12 chess_storage(XMEM)   pDelay,
const complex12 chess_storage(YMEM)   pFilterCoeff,
      int                             numSections
)
```

Donde todos los parámetros se mantienen iguales a excepción de *numSections* que contiene el número de secciones que va a procesar el filtro.

Biquad de una sección con coeficientes escalados

La función *biquad_1ch_bb_1s_step_scale_Process* procesa un canal de datos mediante biquad de una sección igual que la primera función, pero añadiendo la posibilidad de escalar los datos para evitar overflows durante el procesado, para luego devolverlos a su escala inicial. Además, se podrá elegir el desplazamiento tanto en el vector de entrada como en el de salida para leer o guardar los datos.

El filtro realizara entonces los siguiente pasos:

1. A la hora de leer el dato del vector de entrada, éste tiene que haber sido reducido a una escala menor.
2. Al puntero de lectura *in_buff* se le sumará el valor *stepIn* para leer el dato.
3. Se procesará el filtro como en la función *biquad_1ch_bb_1section*
4. El dato resultante se desplazará hasta la escala inicial
5. Al puntero de escritura *out_buff* se le sumará *stepOut* y se escribirá el dato en esa dirección.

La cabecera de la función es:

```
int biquad_1ch_bb_1s_step_scale_Process (
      complex12                     inBuff,
      complex12                     outBuff,
      int                           B,
      complex12 chess_storage(XMEM) pDelay,
const complex12 chess_storage(YMEM) pFilterCoeff,
      int                           stepIn,
      int                           stepOut,
      int                           shift
)
```

Donde los parámetros nuevos son

- *stepIn* es el valor que se suma al puntero de entrada para obtener la siguiente muestra
- *stepOut* es el valor que se suma al puntero de salida para guardar la siguiente muestra procesada
- *shift* es el desplazamiento a la izquierda que se deberá aplicar a los coeficientes a la salida del filtro ya que se da como precondición que los valores han sido desplazados a la derecha *shift* posiciones antes de llamar a la función

Rendimiento

Igual que en el filtro FIR, muestro las tablas de rendimiento del nuevo BiQuad (*complex*) y del anterior (*fix*):

Cuadro 4.7: Rendimiento BiQuad Complex

Función	**Tamaño bloque**	**Ciclos por bloque**
biquad_1ch_bb_1section_Process	Par	$18 + (BlockSize/2) * 17$
	Impar	$29 + (BlockSize/2) * 17$
biquad_1ch_bb_nsection_Process	Par	$13 + ((BlockSize/2) * 16) * NumSections + (NumSections * 12)$
	Impar	$13 + ((BlockSize/2) * 16) * NumSections + (NumSections * 25)$
biquad_1ch_bb_1s_step_scale_Process	Par	$11 + (BlockSize/2) * 16$

De nuevo los resultados parecen dar mejor rendimiento a la antigua función trabajando con tipo *fix*. Y de nuevo la razón es que el trabajo realizado no es equiparable debido a que la función *complex* da por supuesto que se está trabajando con números complejos, luego la operación producto se realiza de forma diferente y trabaja con las dos mitades de cada variable por separado.

Volviendo a la estrategia de comparación utilizada en el filtro FIR, realicé un programa que trabajaba con *fix*, pero de la misma manera que *complex*. Por ello utiliza dos *fix* por cada variable compleja y multiplica de la misma manera que complex.

Cuadro 4.8: Rendimiento BiQuad Fix

Función	**Tamaño bloque**	**Ciclos por bloque**
biquad_1ch_bb_1section_Process	Par	$17 + (BlockSize/2) * 15$
	Impar	$28 + (BlockSize/2) * 15$
biquad_1ch_bb_nsection_Process	Par	$14 + ((BlockSize/2) * 15) * NumSections + (NumSections * 12)$
	Impar	$14 + ((BlockSize/2) * 15) * NumSections + (NumSections * 24)$
biquad_1ch_bb_1s_step_scale_Process	Par	$10 + (BlockSize/2) * 16$

Así, los resultados varían, como se puede ver en la siguiente tabla:

Cuadro 4.9: BiQuad Complex vs BiQuad FixFix

	1 sección par	**2 secciones pares**	**1 sección par escalado mono**
FixFix	3,544,576 (ciclos)	6,967,296	3,646,976
Complex	566,272	1,067,520	529,920
Veces más rápido	*6.26*	*6.53*	*6.88*

Y así vuelve a sobresalir la eficiencia de los nuevos tipos de datos complex.

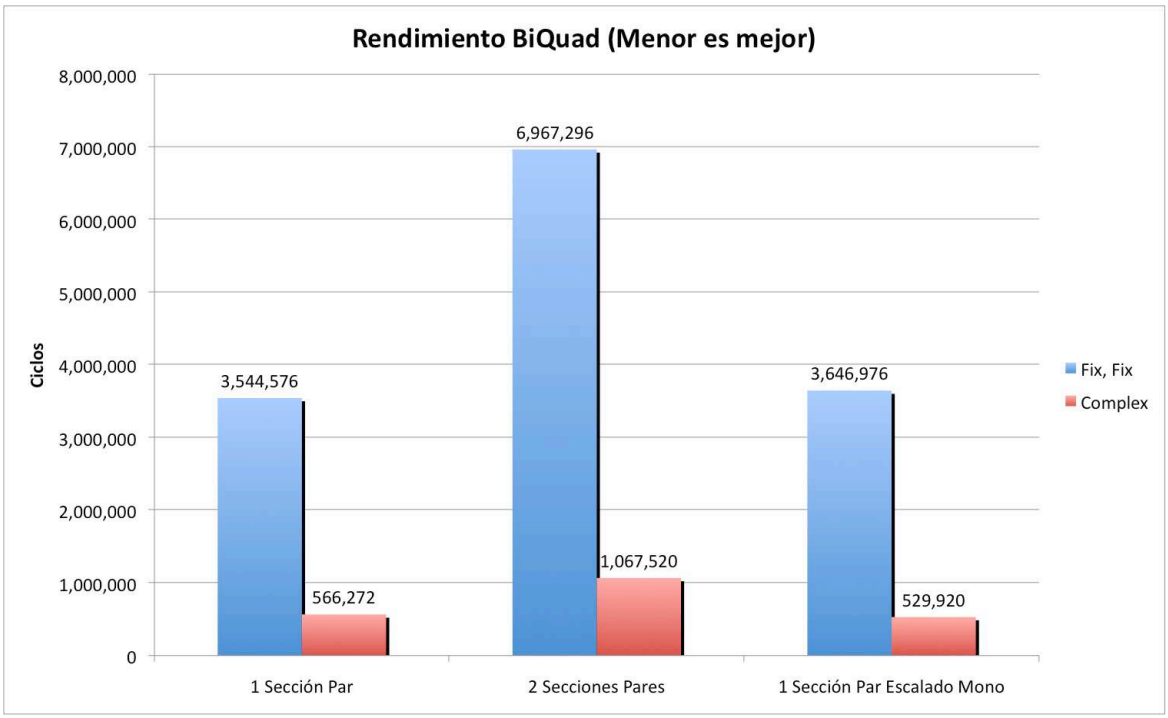

Figura 4.12: BiQuad Complex vs BiQuad FixFix

4.2.4. Filtro Biquad Double Precision Feedback

El filtro *Biquad_dpfb* se utiliza para frecuencias muy bajas donde la precisión del Biquad anterior no es suficiente.

Estuve trabajando solamente una semana con esta función debido a que a la hora de validarlo no fue posible.

Tras haber codificado la nueva versión, siempre se comprueba que los resultados obtenidos en la ejecución WIN32 y CoolFlux sean los mismos. Esto ocurre la gran mayoría de las veces, dado que el código fuente en C es el mismo. La diferencia son los compiladores, y en este caso el compilador de CoolFlux Complex contenía un bug que hacía que los resultados fueran diferentes a la salida.

Cuando el fallo fue descubierto, fue comunicado a nuestro tutor y sería solucionado en la siguiente versión del compilador.

Debido a esto tuve que abandonar este filtro.

4.2.5. Vector Operation

Vector Operation es un conjunto de pequeñas funciones que operan sobre vectores.

Cada operación recibe el tamaño del vector como parámetro.

Las operaciones que he actualizado a *complex* son:

1. *zero_vector_Process*: pone a cero todos los elementos del vector. En el caso de los complex, la parte imaginaria y la real.
2. *copy_vector_Process*: copia un vector en otro pasados ambos como parámetros mediante un puntero.
3. *addvector_Process*: realiza la suma de dos vectores y guarda el resultado en un tercer vector.
4. *subvector_Process*: resta un vector menos otro y guarda el resultado en un tercer vector.

5. *multvector_Process*: multiplica un vector por otro y guarda los resultados redondeados al mismo número de bits que la entrada en un tercer vector.

6. *addconstvector_Process*: suma un número complejo dado a cada elemento de un vector de entrada y almacena el resultado en el otro vector.

7. *multconstvector_Process*: multiplica un número complejo dado a cada elemento de un vector de entrada y almacena el resultado en otro vector.

8. *shiftvector_Process*: desplaza los elementos del vector de entrada tantas posiciones como se pasen por un parámetro entero. Si el parámetro es positivo, el desplazamiento es a la izquierda (multiplica por potencias de dos) y si es negativo desplaza a la derecha (divide).

Además de estas funciones provenientes del proyecto *fix*, he realizado dos funciones nuevas que sólo pueden aplicarse a tipos de datos empaquetados, que son:

1. *join_Process*: dados dos vectores de elementos *fix* y un tercer vector de salida, toma los elementos de un vector como partes reales y los elementos del otro como partes imaginarias. Después, une ambos en una variable *complex12* que almacena en el tercer vector.

2. *unjoin_Process*: realiza el proceso contrario. Dado un vector de elementos *complex12*, extrae las partes imaginaria y real de cada elemento y las guarda en dos vectores *fix*.

```
int join_Process(fix * inbuffR, fix chess_storage(YMEM) * inbuffI, complex12 *outbuff, int B)
{
    int i;
    for (i=0; i<B; i++)
    {
        *outbuff++ = join_complex12(*inbuffR++,*inbuffI++);
    }
    return 0;
}
```

Figura 4.13: Un ejemplo: Código para la función *join_Process*.

Rendimiento

En el cuadro 4.10 se muestra una tabla con los resultados de rendimiento obtenidos en función de los parámetros de las funciones. En esta ocasión, los resultados tanto de la versión nueva (*complex*) como de la antigua (*fix*) fueron los mismos. Esto es debido a que simplemente son operaciones sobre vectores, no es necesario que los productos se realicen de manera compleja para las funciones *fix*.

Cuadro 4.10: Rendimiento Vector Operation Complex

Función	**Ciclos por bloque**
zerovector_Process	$6 + BlockSize$
copyvector_Process	$6 + BlockSize * 2$
addvector_Process	$6 + BlockSize * 2$
subvector_Process	$6 + BlockSize * 2$
multvector_Process	$6 + BlockSize * 2$
addconstvector_Process	$7 + BlockSize * 2$
multconstvector_Process	$6 + BlockSize * 2$
shiftvector_Process	$10 + BlockSize * 2$
join_Process	$6 + BlockSize * 3$
unjoin_Process	$10 + BlockSize * 4$

4.2.6. Complex Fast Fourier Transform (FFT)

Especificación funcional

El módulo de la transformada rápida de Fourier compleja alberga un conjunto de funciones para realizar tanto la transformada como la inversa de la transformada en vectores de entrada complejos.

La longitud de la transformada está restringida a potencias de dos y el tamaño de la transformada está restringido entre 8 y 1024. Todas las transformadas realizan la primera etapa utilizando el modo de direccionamiento bit-reversal[4].

La transformada de N puntos implementa la ecuación siguiente:

$$X(k) = \sum_{n=0}^{N-1} e^{-j2\pi nk/N} * x(n) \tag{4.8}$$

Mientras que la transformada inversa de N puntos implementa la ecuación:

$$x(n) = \frac{1}{N} * \sum_{k=0}^{N-1} e^{j2\pi nk/N} * X(k) \tag{4.9}$$

Estructura de las funciones

En todas las funciones, las dos primeras etapas se combinan en una implementación eficiente en base 4 debido a que se conoce de antemano que sus factores son (1,0) y (0,1). Esta implementación es diferente entre la transformada directa y la inversa.

Después, las etapas 3 a la última están implementadas en base 2, aunque la última etapa de se desenrolla por eficiencia.

[4]Bit-Reversal es una conocida técnica de reordenamiento explícito de bits. Es la permutación de un grupo de bits, como por ejemplo $b_4b_3b_2b_1b_0$, a $b_0b_1b_2b_3b_4$

Funciones El módulo FFT comprende las siguientes funciones:

1. fft_c_noscaling
2. fft_c_stagescaling
3. fft_c_blockscalling
4. fft_c_bfp
5. ifft_c_noscaling
6. ifft_c_stagescaling
7. ifft_c_bfp

fft_c_noscaling

Esta función procesa cada etapa sin realizar ninguna escala. Asume por lo tanto que no puede ocurrir overflow, lo que se deja como responsabilidad de la aplicación.

La cabecera de la función es:

```
int fft_c_noscaling_Process (
   complex12    *Xin,
   complex12    *Xout,
   int           Points,
   FFT_TABLE    *table);
```

Donde

1. *Xin* es el vector de entrada
2. *Xout* es el vector de salida
3. *Points* es el número de puntos de la fft
4. *table* es el puntero a la tabla correcta a utilizar

La función tan sólo realiza la primera etapa; al finalizarla, llama a la función interna *compute_c_inner_stages_noscaling*. Como puede apreciarse en la figura 4.14, que muestra el núcleo de la función, se puede observar que éste reside en un nivel de anidamiento de tres bucles, lo que hace imprescindible la optimización del código generado con el fin de que se ejecute de la forma más eficiente posible. Por ello, se probó con varias técnicas para ello, como el *loop unrolling*[5] de la figura. Se probó con un desenrollado mayor, pero no se obtenía ningún beneficio. Por último, se probó a separar en dos mitades los coeficientes a cargar de memoria, teniendo una mitad en la memoria X y la otra en la memoria Y. Aun así, tampoco se mejoró el rendimiento del núcleo.

fft_c_stagescaling

En esta función, y para prevenir el overflow, la salida de cada etapa es escalada por un factor de dos. Como el número de etapas es $log_2 N$, el factor total de escala será N.

La cabecera de la función es:

```
int fft_c_stagescaling_Process (
   complex12  *Xin,
   complex12  *Xout,
   int         Points,
   FFT_TABLE  *table);
```

Los parámetros son los mismos que en la función anterior.

Esta función llama, tras realizar el primer escalado y la primera etapa, a la función *compute_c_inner_stages_stagescaling*.

[5]En español, *desenrollado de bucles*, es una técnica de programación para optimizar partes de código. La idea es reducir el número de iteraciones que realiza un bucle, y con ello la ejecución de instrucciones de *overhead*. Para conseguirlo, se replica el código dentro del bucle actualizando los contadores correctamente para realizar, por ejemplo, la mitad de iteraciones.

```
for (stage = 3; stage <= nr_stages; stage++)   chess_loop_range(1,)
    {
        ...

        for(i=0; i < num_groups; i++) chess_loop_range(1,)
        {
            ...

            for (j = 0; j < num_butterflies/2; j++)    chess_loop_range(2,) chess_prepare_for_pipelining
            {
                x1 = (*baseX1++);
                x3 = (*baseX1--);
                x2 = (*baseX2_rd++);
                x4 = (*baseX2_rd++);

                w_current = w[k];
                k += index*wfactor;
                w_current1 = w[k];
                k += index*wfactor;

                a1 = x1 + (x2 * w_current);
                a2 = x1 - (x2 * w_current);
                a3 = x3 + (x4 * w_current1);
                a4 = x3 - (x4 * w_current1);

                *baseX1++ = (complex12)(a1);
                *baseX1++ = (complex12)(a3);
                *baseX2_wr++ = (complex12)(a2);
                *baseX2_wr++ = (complex12)(a4);
            }

            ...

        }

        ...

    }
```

Figura 4.14: Código del núcleo de la función *fft_c_noscaling* calculando dos mariposas por vuelta.

fft_c_blockscaling

En esta función, las entradas de la fft primero se escalan en un factor determinado por la aplicación y, en la última escala, se pasan a la función. El valor de escala siempre tiene que ser potencia de dos.

La idea es que la aplicación determine el rango de la entrada de forma dinámica, y si ya es menor que el rango admitido, se llama a la función de *block-scaling* con un valor aplicable, para no tener que llamar a la versión de *stage-scaling*, porque esta última realizaría una escala demasiado extensa, perdiendo precisión.

La cabecera de la función es:

```
int fft_c_blockscaling_Process (
   complex12  *Xin
```

```
   complex12  *Xout,
   int         Points,
   FFT_TABLE  *table,
   int         scale);
```

Donde

1. *scale* es la escala que va a ser aplicada a la entrada

Esta función también realiza sólo la primera etapa de la transformada y su escalado, para luego llamar a la función interna *compute_c_inner_stages_noscaling*.

fft_c_bfp

En este caso, en cada etapa, la función determina si realizar la escala es necesario (dividiendo por dos) para evitar el overflow en función de los valores de entrada.

Si es necesario, trabaja como la función *fft_c_stagescaling*; en otro caso, trabajará como *fft_c_noscaling*.

En el parámetro *scale* devuelve el desplazamiento a la izquierda necesario para recuperar los valores de salida a escala original.

La cabecera de la función es:

```
int fft_c_bfp_Process (
   complex12  *Xin,
   complex12  *Xout,
   int         Points,
   FFT_TABLE  *table,
   int        &scale);
```

Donde *scale* se pasa como referencia, como en C++, que está permitido también en el compilador de CoolFlux.

ifft_c_noscaling

Primera de las funciones que realiza la inversa de la transformada. No realiza ninguna escala, lo que la hace muy útil para comprobar si la transformada directa se ha hecho correctamente, ya que después de aplicar las dos funciones en serie, los datos obtenidos son los originales.

La cabecera de la función es:

```
int ifft_c_noscaling_Process (
   complex12  *Xin,
   complex12  *Xout,
   int         Points,
   FFT_TABLE  *itable);
```

Esta función también llama internamente a la función *compute_c_inner_stages_noscaling*.

ifft_c_stagescaling

Al igual que su homóloga directa, esta función realiza el escalado escalando entre dos después de cada etapa.

La cabecera de la función es:

```
int ifft_c_stagescaling_Process (
   complex12   *Xin,
   complex12   *Xout,
   int          Points,
   FFT_TABLE   *itable);
```

La función también llama internamente a *compute_c_inner_stages_stagescaling*.

ifft_c_bfp

Esta función realiza el mismo trabajo extra que su homóloga directa, la *fft_c_bfp*.

La cabecera de la función es:

```
int ifft_c_bfp_Process (
   complex12  *Xin,
   complex12  *Xout,
   int         Points,
   FFT_TABLE  *table,
   int        &scale);
```

La función llama internamente a *compute_inner_stages_bfp*.

Rendimiento

A continuación, muestro una tabla de resultados finales al ejecutar una prueba con distintos tamaños de muestra y tabla de coeficientes. En cada fila se muestran alternativamente los resultados antiguos (*fix*) y a continuación los nuevos (*complex*):

Cuadro 4.11: Rendimiento FFT

Tamaño de la FFT	**fft_c_noscaling**	**fft_c_stagescaling**	**fft_c_bfp**
8 (fix)	171	173	307
8 (complex)	111	105	189
64 (fix)	1800	1912	4193
64 (complex)	1157	1148	2304
256 (fix)	9698	9634	21521
256 (complex)	5933	5922	11868
2048 (fix)	94945	103649	232241
2048 (complex)	65169	65155	130361

En este caso, el *speedup* conseguido oscila entre 1.45 y 1.82. Estas cifras no son tan buenas como las de las funciones anteriores, pero el motivo se explicará en el capítulo siguiente. A continuación muestro los resultados en modo gráfico. Las tablas tratan distintos tamaños de muestras para que se pueda comprobar que los resultados son siempre proporcionales.

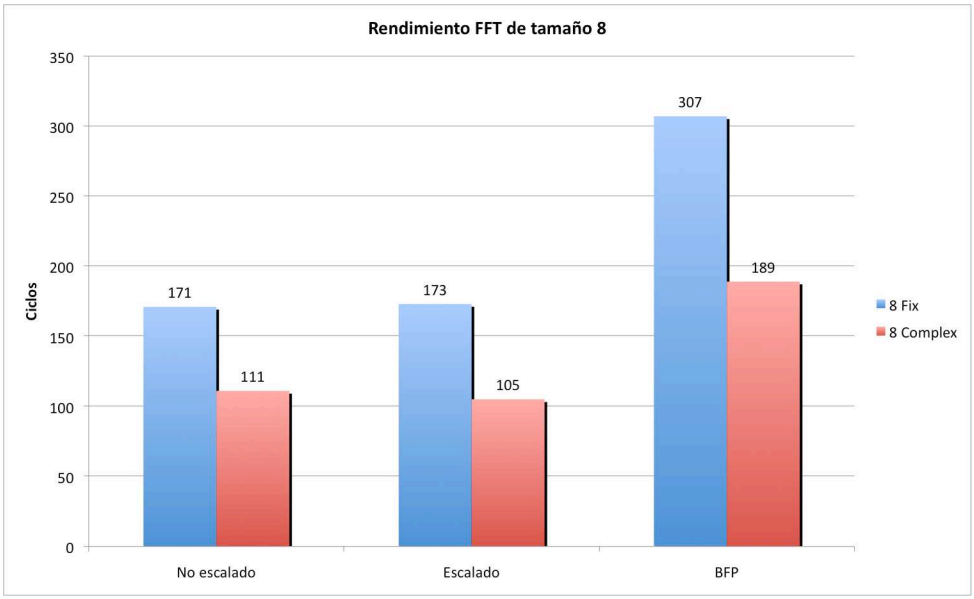

Figura 4.15: FFT Complex 8 vs FFT FixFix 8

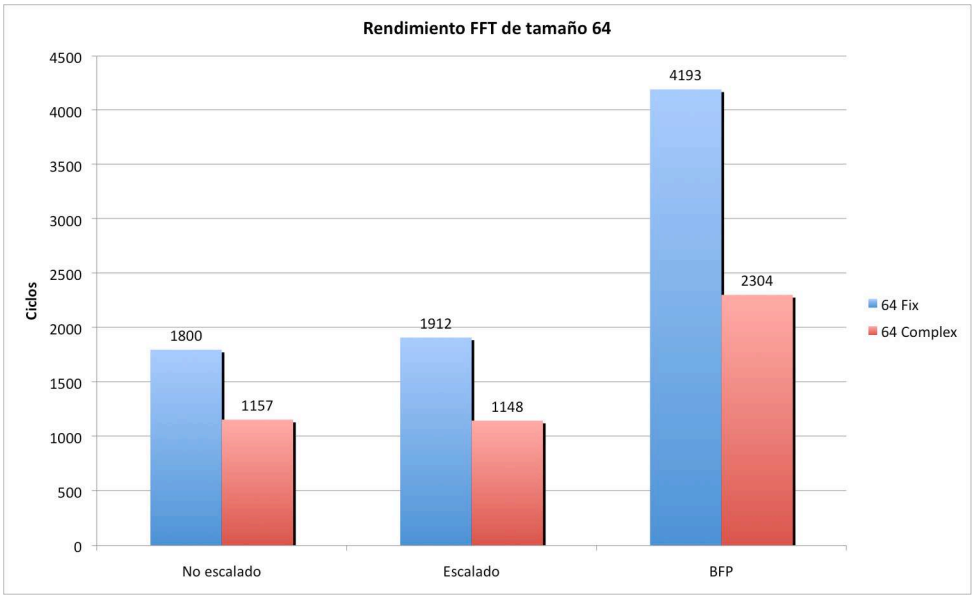

Figura 4.16: FFT Complex 64 vs FFT FixFix 64

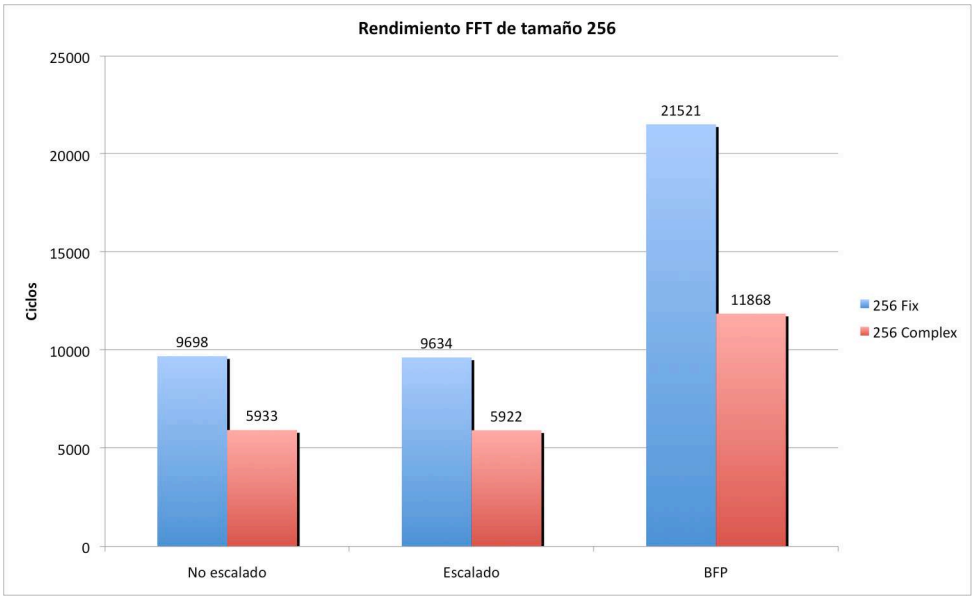

Figura 4.17: FFT Complex 256 vs FFT FixFix 256

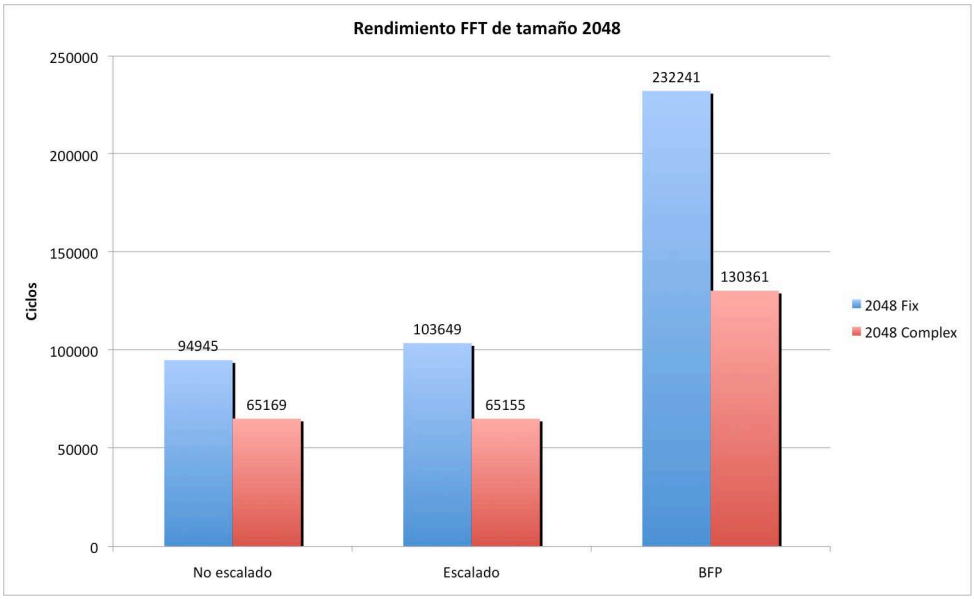

Figura 4.18: FFT Complex 2048 vs FFT FixFix 2048

Capítulo 5

Conclusiones

Con estas nuevas librerías el procesador aumentará su eficiencia y productividad, lo que permitirá realizar más trabajo en menos tiempo. La productividad es una característica esencial en el procesador de menor consumo del mercado hasta la fecha, ya que su ámbito de trabajo está en reproductores portátiles, teléfonos móviles o implantes auditivos, los cuales necesitan que la batería dure lo máximo posible. Este hecho hace que la arquitectura del CoolFlux Complex tenga que ser muy sencilla, lo que ha repercutido en gran medida en los resultados obtenidos si se quiere comparar con otros procesadores de mayor rendimiento. Por ejemplo, la serie mejorada 16xxx de DSP de Lucent contiene ocho acumuladores, mientras que el CoolFlux Complex cuenta con la mitad; otro ejemplo sería la imposibilidad de utilizar instrucciones VLIW[1], que ayudan en gran medida a paralelizar trabajo en caso de que el compilador sea efectivo.

A lo largo de todo el proceso de optimización se han conseguido resultados parecidos. Primeramente, al comparar los nuevos ciclos de ejecución con los antiguos, los resultados decepcionaban ya que eran ligeramente peores. Seguidamente se procedía a realizar una prueba que fuera equitativa al trabajo que realizaban las nuevas funciones y entonces el número de ciclos de las antiguas funciones crecía drásticamente. Con tan sólo esas comprobaciones, ya era suficiente mejora como para incorporar las nuevas funciones a la biblioteca general de librerías del nuevo procesador CoolFlux Complex.

[1]Del inglés, Very Long Instruction Word (palabra de instrucción muy larga): un número fijo de instrucciones (4-8) se agrupan para formar un paquete o palabra de instrucción larga, planificadas estáticamente por el compilador.

De todas formas, los resultados aún no son óptimos. Dado que el compilador utilizado todavía no está totalmente optimizado para los nuevos tipos de datos *complex*, hay secciones de código que al ser pasadas de código C a código ensamblador, no aprovechan aún toda la capacidad del procesador, es decir, aparecen las temidas *nops* (no-operation). Las nops son huecos en el desarrollo de una ejecución en los que el procesador no realiza trabajo útil porque no tiene la información suficiente para ello o los componentes que necesita utilizar están en uso. Si estos huecos aparecen fuera de cualquier bucle, el problema no es muy importante. Pero si aparecen en núcleos de algoritmos que se repiten miles de veces, el problema aumenta.

En mi caso, precisamente esto ocurrió en el proyecto de la Transformada Rápida de Fourier (FFT). La parte de código más interna, en un anidamiento de bucles de tres niveles, era compilada con varias *nops*, lo que derivó en los resultados antes mostrados. Sin esas *nops*, seguramente el *speedup* conseguido habría sido entre 2 y 4, ya que esa pequeña parte de código se ejecutaría miles de veces para tamaños de tabla de uso real.

En cuanto a su facilidad de uso, debido a que las librerías *complex* están basadas en las anteriores *fix*, cualquier programador del procesador que las haya utilizado anteriormente no encontrará ningún problema. Tan sólo deberá cambiar los tipos de datos para aprovechar que cada variable alberga tanto la parte real como la imaginaria de cada número complejo.

Como continuación de este proyecto, ahora que las funciones ya están hechas, lo que sigue será la optimización del compilador para poder aprovechar estos nuevos tipos de datos con la máxima eficiencia. Por otro lado, siempre cabe la posibilidad de optimizar todavía más el código hecho, así como su mantenimiento y/o actualización para siguientes versiones del procesador. En cuanto al CoolFlux Complex, se está trabajando ya en un diseño nuevo de éste para soportar operaciones en coma flotante. Además, también se está estudiando modificarlo para que admita una mayor precisión en los datos, con más bits por palabra.

Bibliografía

[1] *CoolFlux DSP, the embedded ultra low power C-programmable DSP core*,
versión 3.5,
NXP.

[2] *CoolFlux DSP GSPX*,
Hans Roeven, Jeroen Coninx, Marleen Ade
versión 2004,
NXP.

[3] *CoolFlux Complex DSP Assembly Manual*,
versión 0.1 de Febrero 2007,
NXP.

[4] *CoolFlux Complex DSP C Manual*,
versión 0.2 de Marzo 2007,
NXP.

[5] *The Scientist and Engineer's Guide to Digital Processing*
Steven W. Smith
California Technical Publishing

[6] *Discrete-Time Signal Processing*,
Alan V.Oppenheim, Ronald W. Schafer
Editorial Prentice Hall.

[7] *DSP Processor Fundamentals Architectures and Features*,
Phil Lapsley, Jeff Bier, Amit Shoham, Edward A. Lee
IEEE Press,
Editorial Board.

[8] *Transparencias de la asignatura "Arquitecturas Especializadas: Procesadores DSP de alto rendimiento"*,
Luis Piñuel Moreno

Departamento de Arquitectura de Computadores y Automática,
Universidad Complutense de Madrid.